Pip: Detecting the Unexpected in Distributed Systems

Patrick Reynolds*, Charles Killian†, Janet L. Wiener‡,
Jeffrey C. Mogul‡, Mehul A. Shah‡, and Amin Vahdat†

*Duke University † UC San Diego ‡ HP Labs, Palo Alto

Abstract

Bugs in distributed systems are often hard to find. Many bugs reflect discrepancies between a system's behavior and the programmer's assumptions about that behavior. We present Pip[1], an infrastructure for comparing actual behavior and expected behavior to expose structural errors and performance problems in distributed systems. Pip allows programmers to express, in a declarative language, expectations about the system's communications structure, timing, and resource consumption. Pip includes system instrumentation and annotation tools to log actual system behavior, and visualization and query tools for exploring expected and unexpected behavior[2]. Pip allows a developer to quickly understand and debug both familiar and unfamiliar systems.

We applied Pip to several applications, including FAB, SplitStream, Bullet, and RanSub. We generated most of the instrumentation for all four applications automatically. We found the needed expectations easy to write, starting in each case with automatically generated expectations. Pip found unexpected behavior in each application, and helped to isolate the causes of poor performance and incorrect behavior.

1 Introduction

Distributed systems exhibit more complex behavior than applications running on a single node. For instance, a single logical operation may touch dozens of nodes and send hundreds of messages. Distributed behavior is also more varied, because the placement and order of events can differ from one operation to the next. Bugs in distributed systems are therefore hard to find, because they may affect or depend on many nodes or specific sequences of behavior.

In this paper, we present Pip, a system for automatically checking the behavior of a distributed system against a programmer's expectations about the system. Pip classifies system behaviors as valid or invalid, groups behaviors into sets that can be reasoned about, and presents overall behavior in several forms suited to discovering or verifying the correctness of system behavior.

Bugs in distributed systems can affect structure, performance, or both. A structural bug results in processing or communication happening at the wrong place or in the wrong order. A performance bug results in processing taking too much or too little of any important resource. For example, a request that takes too long may indicate a bottleneck, while a request that finishes too quickly may indicate truncated processing or some other error. Pip supports expressing expectations about both structure and performance and so can find a wide variety of bugs.

We wrote Pip for three broad types of users:

- original developers, verifying or debugging their own system;
- secondary developers, learning about an existing system; and
- system maintainers, monitoring a system for changes.

Our experience shows three major benefits of Pip. First, expectations are a simple and flexible way to express system behavior. Second, automatically checking expectations helps users find bugs that other approaches would not find or would not find as easily. Finally, the combination of expectations and visualization helps programmers explore and learn about unfamiliar systems.

1.1 Context

Programmers employ a variety of techniques for debugging distributed systems. Pip complements existing approaches, targeting different types of systems or different types of bugs. Table 1 shows four approaches and the types of systems or bugs for which they are most useful.

Traditional debuggers and profilers like gdb and gprof are mature and powerful tools for low-level bugs. However, gdb applies to only one node at a time and generally requires execution to be paused for examination. Gprof produces results that can be aggregated offline but has no support for tracing large-scale operations through the network. It is more useful for tuning small blocks of code than distributed algorithms and their emergent behavior.

More recent tools such as Project 5 [1], Magpie [2], and Pinpoint [5] infer causal paths based on traces of net-

www.manaraa.com

Approach	Scenario
gdb and gprof	low-level bugs well illustrated by a single node; core dumps
black boxes	systems with no source-code access, enough self-consistency for statistical inference
model checking	small systems with difficult-to-reproduce bugs
printf	bugs detectable with simple, localized log analyses

Table 1: Other techniques for debugging distributed systems.

work, application, or OS events. Project 5 merely reports inferred behavior, while Magpie and Pinpoint cluster similar behavior and suggest outliers as possible indicators of bugs. Pip also uses causal paths, but instead of relying on statistics and inference, Pip uses explicit path identifiers and programmer-written expectations to gather and check program behavior. We discuss the relationship between Pip and other causal path debugging systems further in Section 6.

Programmers may find some bugs using model checking [10, 16]. Model checking is exhaustive, covering all possible behaviors, while Pip and all the other techniques mentioned above check only the behaviors exhibited in actual runs of the system. However, model checking is expensive and is practically limited to small systems and short runs—often just tens of events. Model checking is often applied to specifications, leaving a system like Pip to check the correctness of the implementation. Finally, unlike model checking, Pip can check performance characteristics.

In practice, the dominant tool for debugging distributed systems has remained unchanged for over twenty years: printf to log files. The programmer analyzes the resulting log files manually or with application-specific validators written in a scripting or string-processing language. In our experience, incautious addition of logging statements generates too many events, effectively burying the few events that indicate or explain actual bugs.

Debugging with log files is feasible when bugs are apparent from a small number of nearby events. If a single invariant is violated, a log file may reveal the violation and a few events that preceded it. However, finding correctness or performance problems in a distributed system of any scale is incredibly labor intensive. In our own experience, it can take days to track down seemingly simple errors. Further, scripts to check log files are brittle because they do not separate the programmer's expectations from the code that checks them, and they must be written anew for each system and for each property being checked.

1.2 Contributions and results

Pip makes the following contributions:

- An expectations language for writing concise, declarative descriptions of the expected behavior of large distributed systems. We present our language design, along with design principles for handling parallelism and for balancing over- and under-constraint of system behavior.
- A set of tools for gathering events, checking behavior, and visualizing valid and invalid behaviors.
- Tools to generate expectations automatically from system traces. These expectations are often more concise and readable than any other summary of system behavior, and bugs can be obvious just from reading them.

We applied Pip to several distributed systems, including FAB [25], SplitStream [4], Bullet [13, 15], and RanSub [14]. Pip automatically generated most of the instrumentation for all four applications. We wrote expectations to uncover unexpected behavior, starting in each case from automatically generated expectations. Pip found unexpected behavior in each application and helped to isolate the causes of poor performance and incorrect behavior.

The rest of this paper is organized as follows. Section 2 contains an overview of the Pip architecture and tool chain. Sections 3 and 4 describe in detail the design and implementation of our expectation language and annotation system, respectively. Section 5 describes our results.

2 Architecture

Pip traces the behavior of a running application, checks that behavior against programmer expectations, and displays the resulting valid and invalid behavior in a GUI using several different visualizations.

2.1 Behavior model

We define a model of application behavior for use with Pip. This model does not cover every possible application, but we found it natural for the systems we analyzed.

The basic unit of application behavior in Pip is a path instance. Path instances are often causal and are often in response to an outside input such as a user request. A path instance includes events on one or more hosts and can include events that occur in parallel. In a distributed file system, a path instance might be a block read, a write, or a data migration. In a three-tier web service, path instances might occur in response to user requests. Pip allows the programmer to define paths in whatever way is appropriate for the system being debugged.

المنارة للاستشارات
www.manaraa.com

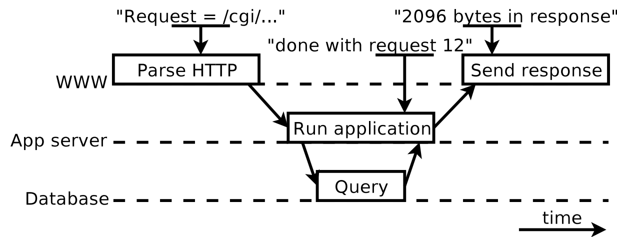

Figure 1: A sample causal path from a three-tier system.

Each path instance is an ordered series of timestamped events. The Pip model defines three types of events: tasks, messages, and notices. A *task* is like a profiled procedure call: an interval of processing with a beginning and an end, and measurements of resources consumed. Tasks may nest inside other tasks but otherwise may not overlap other tasks on the same thread. Tasks may include asynchronous events like timer callbacks, which Pip normally associates with the path instances that scheduled them. A *message* is any communication event between hosts or threads, whether a network message, a lock, or a timer. Pip records messages when they are sent and again when they are received. Finally, a *notice* is an opaque string—like a log message, with a timestamp and a path identifier for context.

Figure 1 shows a sample path instance. Each dashed horizontal line indicates one host, with time proceeding to the right. The boxes are tasks, which run on a single host from a start time to an end time. The diagonal arrows are messages sent from one host to another. The labels in quotation marks are notices, which occur at one instant on a host.

Pip associates each recorded event with a thread. An event-handling system that dispatches related events to several different threads will be treated as having one logical thread. Thus, two path instances that differ only on which threads they are dispatched will appear to have identical behavior.

Our choice of tasks, messages, and notices is well suited to a wide range of distributed applications. Tasks correspond to subroutines that do significant processing. In an event-based system, tasks can correspond to event-handling routines. Messages correspond to network communication, locks, and timers. Notices capture many other types of decisions or events an application might wish to record.

2.2 Tool chain

Pip is a suite of programs that work together to gather, check, and display the behavior of distributed systems. Figure 2 shows the workflow for a programmer using Pip. Each step is described in more detail below.

Annotated applications: Programs linked against Pip's annotation library generate events and resource measurements as they run. Pip logs these events into trace files, one per kernel-level thread on each host. We optimized the annotation library for efficiency and low memory overhead; it performs no analysis while the application is running.

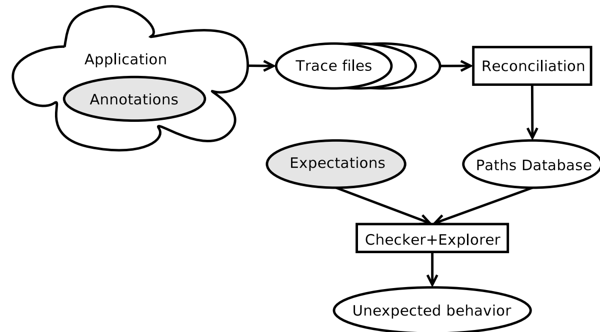

Figure 2: Pip workflow. Shaded ovals represent input that must be at least partially written by the programmer.

We found that the required annotations are easiest to add when communication, event handling, and logging are handled by specialized components or by a supported middleware library. Such concentration is common in large-scale distributed systems. For applications linked against a supported middleware library, a modified version of the library can generate automatic annotations for every network message, remote procedure call, and network-event handler. Programmers can add more annotations to anything not annotated automatically.

A separate program gathers traces from each host and reconciles them. Reconciliation includes pairing message send and receive events, pairing task start and end events, and performing a few sanity checks. Reconciliation writes events to a database as a series of path instances. Normally, reconciliation is run offline, parsing log files from a short test run. However, Pip may also be run in an online mode, adding paths to the database and checking them as soon as they complete. Section 4 describes annotations and reconciliation in more detail.

Expectations: Programmers write an external description of expected program behavior. The expectations take two forms: *recognizers*, which validate or invalidate individual path instances, and *aggregates*, which assert properties of sets of path instances. Pip can generate initial recognizers automatically, based on recorded program behavior. These generated recognizers serve as a concise, readable description of actual program behavior. Section 3 describes expectations in more detail.

Formally, a set of recognizers in Pip is a grammar, defining valid and invalid sequences of events. In its current form, Pip allows users to define non-deterministic finite-state machines to check a regular grammar. We chose to define a domain-specific language for defining

www.manaraa.com

these grammars because our language more closely mirrors how programmers reason about behavior in their applications. We believe this choice simplifies writing and maintaining expectations.

Expectation checker: If the programmer provides any expectations, Pip checks all traced behavior against them. These checks can be done non-interactively, to generate a list of violations, or they can be incorporated into the behavior explorer (below). Section 3.5 describes the implementation and performance of expectation checking.

The expectation violations that Pip uncovers do not always indicate bugs in the system being tested. Sometimes, the errors are in the expectations or in the annotations. Using Pip entails changing the application, the expectations, and the annotations until no further unexpected behavior is found. Unexpected paths due to incorrect expectations or annotations can loosely be called *false positives*, though they are not due to any incorrect inference by Pip.

Behavior explorer: Pip provides an interactive GUI environment that displays causal structure, communication structure, sets of validated and invalidated paths, and resource graphs for tasks or paths. Even without writing any expectations, programmers can visualize most aspects of application behavior. Pip stores all of its paths in an SQL database so that users can explore and check application behavior in ways that Pip may not support directly. Space constraints prevent us from describing the GUI or the database schema further here.

3 Expectations

Both checking and visualization in Pip start with expectations. Using Pip's declarative expectations language, programmers can describe their intentions about a system's structure, timing, and resource consumption.

3.1 Design considerations

Our goal is to provide a declarative, domain-specific expectations language that is more expressive than general-purpose languages, resulting in expectations that are easier to write and maintain. Programmers using Pip should be able to find more complex bugs with less effort than programmers checking behavior with scripts or programs written in general-purpose languages.

With expressiveness in mind, we present three goals for any expectations language:

1. Expectations written in the language must accept all valid paths. One recognizer should be able to accept a whole family of paths—e.g., all read operations in a distributed file system or all CGI page loads in a webserver—even if they vary slightly. In some systems, particularly event-driven systems, the order of events might vary from one path instance to the next.
2. Expectations written in the language must reject as many invalid paths as possible. The language should allow the programmer to be as specific as possible about task placement, event order, and communication patterns, so that any deviations can be categorized as unexpected behavior.
3. The language should make simple expectations easy to express.

We designed Pip with several real systems in mind: peer-to-peer systems, multicast protocols, distributed file systems, and three-tier web servers, among others. Pip also draws inspiration from two platforms for building distributed systems: Mace[3] [12] and SEDA [27]. The result is that Pip supports thread-oriented systems, event-handling systems, and hybrids. We gave special consideration to event-handling systems that dispatch events to multiple threads in a pool, i.e., for multiprocessors or to allow blocking code in event handlers.

3.2 Approaches to parallelism

The key difficulty in designing an expectations language is expressing parallelism. Parallelism in distributed systems originates from three main sources: hosts, threads, and event handlers. Processing happens in parallel on different hosts or on different threads within the same host, either with or without synchronization. Event-based systems may exhibit additional parallelism if events arrive in an unknown order.

Pip first reduces the parallelism apparent in an application by dividing behavior into paths. Although a path may or may not have internal parallelism, a person writing Pip expectations is shielded from the complexity of matching complex interleavings of many paths at once.

Pip organizes the parallelism within a path into threads. The threads primitive applies whether two threads are on the same host or on different hosts. Pip's expectation language exposes threading by allowing programmers to write *thread patterns*, which recognize the behavior of one or more threads in the same path instance.

Even within a thread, application behavior can be nondeterministic. Applications with multiple sources of events (e.g., timers or network sockets) might not always process events in the same order. Thus, Pip allows programmers to write *futures*, which are sequences of events that happen at any time after their declaration.

One early design for Pip's expectation language treated all events on all hosts as a single, logical thread. There were no thread patterns to match parallel behavior. This paradigm worked well for distributed hash tables (DHTs) and three-tier systems, in which paths are largely linear, with processing across threads or hosts se-

www.manaraa.com

```
// Read3Others is a validating recognizer
validator Read3Others {
    // no voluntary context switches: never block
    limit(VOL_CS, 0);
    // one Client, issues a read request to Coordinator
    thread Client(*, 1) {
        send(Coordinator) limit(SIZE, {=44b}); // exactly 44 bytes
        recv(Coordinator); }
    // one Coordinator, requests blocks from three Peers
    thread Coordinator(*, 1) {
        recv(Client) limit(SIZE, {=44b});
        task("fabrpc::Read") {
            repeat 3 { send(Peer); }
            repeat 2 {
                recv(Peer);
                task("quorumrpc::ReadReply"); }
            future { // these statements match events now or later
                recv(Peer);
                task("quorumrpc::ReadReply"); } }
        send(Client); }
    // exactly three Peers, respond to Coordinator
    thread Peer(*, 3) {
        recv(Coordinator);
        task("quorumrpc::ReadReq") { send(Coordinator); } } }
// "assert" indicates an aggregate expectation
assert(average(REAL_TIME, Read3Others) < 30ms);
```

Figure 3: FAB read protocol, expressed as an expectation.

rialized. It worked poorly, however, for multicast protocols, distributed file systems, and other systems where a single path might be active on two hosts or threads at the same time. We tried a `split` keyword to allow behavior to occur in parallel on multiple threads or hosts, but it was awkward and could not describe systems with varying degrees of parallelism. The current design, using thread patterns and futures, can naturally express a wider variety of distributed systems.

3.3 Expectation language description

Pip defines two types of expectations: *recognizers* and *aggregates*. A recognizer is a description of structural and performance behavior. Each recognizer classifies a given path instance as matching, matching with performance violations, or non-matching. Aggregates are assertions about properties of sets of path instances. For example, an aggregate might state that a specific number of path instances must match a given recognizer, or that the average or 95th percentile CPU time consumed by a set of path instances must be below some threshold.

Figure 3 shows a recognizer and an aggregate expectation describing common read events in FAB [25], a distributed block-storage system. The `limit` statements are optional and are often omitted in real recognizers. They are included here for illustration.

FAB read events have five threads: one client, one I/O coordinator, and three peers storing replicas of the requested block. Because FAB reads follow a quorum protocol, the coordinator sends three read requests but only needs two replies before it can return the block to

```
validator fab_109 {
    thread t_7(*, 1) {
        send(t_9); recv(t_9); }
    thread t_9(*, 1) {
        recv(t_7);
        task("fabrpc::Read") {
            send(t_1);
            send(t_1);
            send(t_1);
            recv(t_1);
            task("quorumrpc::ReadReply");
            recv(t_1);
            task("quorumrpc::ReadReply"); }
        send(t_7);
        recv(t_1);
        task("quorumrpc::ReadReply"); }
    thread t_1(*, 3) {
        recv(t_9);
        task("quorumrpc::ReadReq") { send(t_9); } } }
```

Figure 4: Automatically generated expectation for the FAB read protocol, from which we derived the expectation in Figure 3.

the client. The final read reply may happen before or after the coordinator sends the newly read block to the client. Figure 4 shows a recognizer generated automatically from a trace of FAB, from which we derived the recognizer in Figure 3.

The recognizer in Figure 3 matches only a 2-of-3 quorum, even though FAB can handle other degrees of replication. Recognizers for other quorum sizes differ only by constants. Similarly, recognizers for other systems might depend on deployment-specific parameters, such as the number of hosts, network latencies, or the desired depth of a multicast tree. In all cases, recognizers for different sizes or speeds vary only by one or a few constants. Pip could be extended to allow parameterized recognizers, which would simplify the maintenance of expectations for systems with multiple, different deployments.

Pip currently provides no easy way to constrain similar behavior. For example, if two loops must execute the same number of times or if communication must go to and from the same host, Pip provides no means to say so. Variables would allow an expectations writer to define one section of behavior in terms of a previously observed section. Variables are also a natural way to implement parameterized recognizers, as described above.

The following sections describe the syntax of recognizers and aggregate expectations.

3.3.1 Recognizers

Each recognizer can be a *validator*, an *invalidator*, or a building block for other expectations. A path instance is considered valid behavior if it matches at least one validator and no invalidators. Ideally, the validators in an expectations file describe *all* expected behavior in a system, so any unmatched path instances imply invalid be-

www.manaraa.com

havior. Invalidators may be used to indicate exceptions to validators, or as a simple way to check for specific bugs that the programmer knows about in advance.

Each recognizer can match either complete path instances or fragments. A *complete recognizer* must describe all behavior in a path instance, while a *fragment recognizer* can match any contiguous part of a path instance. Fragment recognizers are often, but not always, invalidators, recognizing short sequences of events that invalidate an entire path. The validator/invalidator and complete/fragment designations are orthogonal.

A recognizer matches path instances much the same way a regular expression matches character strings. A complete recognizer is similar to a regular expression that is constrained to match entire strings. Pip's recognizers define regular languages, and the expectation checker approximates a finite state machine.

Each recognizer in Pip consists of expectation statements. Each statement can be a literal, matching exactly one event in a path instance; a variant, matching zero or more events in a path instance; a future, matching a block of events now or later; or a limit, constraining resource consumption. What follows is a description of the expectation statements used in Pip. Most of these statements are illustrated in Figure 3.

Thread patterns: Path instances in Pip consist of one or more threads or thread pools, depending on system organization. There must be at least one thread per host participating in the path. All complete (not fragment) recognizers consist of thread patterns, each of which matches threads. A whole path instance matches a recognizer if each thread matches a thread pattern. Pip's syntax for a thread pattern is:

thread(*where*, *count*) {*statements*}

Where is a hostname, or "*" to match any host. *Count* is the number of threads allowed to match, or an allowable range. *Statements* is a block of expectation statements.

Literal statements: Literal expectation statements correspond exactly to the types of path events described in Section 2. The four types of literal expectation statements are `task`, `notice`, `send`, and `recv`.

A `task` statement matches a single task event and any nested events in a path instance. The syntax is:

task(*name*) {*statements*}

Name is a string or regular expression to match the task event's name. The optional *statements* block contains zero or more statements to match recursively against the task event's subtasks, notices, and messages.

A `notice` statement matches a single notice event. `Notice` statements take a string or regular expression to match against the text of the notice event.

`Send` and `recv` statements match the endpoints of a single message event. Both statements take an identifier indicating which thread pattern or which node the message is going to or arriving from.

Variant statements: Variant expectation components specify a fragment that can match zero or more actual events in a path instance. The five types of variant statements are `repeat`, `maybe`, `xor`, `any`, and `include`.

A `repeat` statement indicates that a given block of code will be repeated n times, for n in a given range. The `maybe` statement is a shortcut for `repeat between 0 and 1`. The syntax of `repeat` and `maybe` is:

repeat between *low* **and** *high* { *statements* }
maybe { *statements* }

An `xor` statement indicates that exactly one of the stated branches will occur. The syntax of `xor` is:

xor {
  branch: *statements*
  branch: *statements*
  ... *(any number of branch statements)*
}

An `any` statement matches zero or more path events of any type. An `any` statement is equivalent to ".*" in a regular expression, allowing an expectation writer to avoid explicitly matching a sequence of uninteresting events.

An `include` statement includes a fragment expectation inline as a macro expansion. The `include` statement improves readability and reduces the need to copy and paste code.

Futures: Some systems, particularly event-handling systems, can allow the order and number of events to vary from one path instance to the next. Pip accommodates this fact using `future` statements and optional `done` statements. The syntax for `future` and `done` statements is:

future [*name*] {*statements*}
done(*name*);

A `future` statement indicates that the associated block of statements will match contiguously and in order at or after the current point in the path instance. Loosely, a future states that something will happen either now or later. Futures may be nested: when one future encloses another, it means that the outer one must match before the inner one. Futures may also be nested in (or may include) variant statements. Futures are useful for imposing partial ordering of events, including asynchronous events. Specifying several futures in a row indicates a set of events that may finish in any order. The recognizer in Figure 3 uses futures to recognize a 2-of-3 quorum in FAB: two peers must respond immediately, while the third may reply at any later time.

www.manaraa.com

A `done` statement indicates that events described by a given future statement (identified by its name) must match prior to the point of the `done` statement. All futures must match by the end of the path instance, with or without a `done` statement, or else the recognizer does not match the path instance.

Limits: Programmers can express upper and lower limits on the resources that any task, message, or path can consume. Pip defines several metrics, including real time, CPU time, number of context switches, and message size and latency (the only metrics that apply to messages). A limit on the CPU time of a path is evaluated against the sum of the CPU times of all the tasks on that path. A limit on the real time of a path is evaluated based on the time between the first and last events on the path.

Recognizer sets: One recognizer may be defined in terms of other recognizers. For example, recognizer C may be defined as matching any path instance that matches A and does not match B, or the set difference $A - B$.

3.3.2 Aggregates

Recognizers organize path instances into sets. Aggregate expectations allow programmers to reason about the properties of those sets. Pip defines functions that return properties of sets, including:

- `instances` returns the number of instances matched by a given recognizer.
- `min`, `max`, `avg`, and `stddev` return the minimum, maximum, average, and standard deviation of the path instances' consumption of any resource.

Aggregate expectations are assertions defined in terms of these functions. Pip supports common arithmetic and comparative operators, as well as simple functions like logarithms and exponents. For example:

assert(**average**(CPU_TIME, ReadOperation) $<$ 0.5s);

This statement is true if the average CPU time consumed by a path instance matching the ReadOperation recognizer is less than 0.5 seconds.

3.4 Avoiding over- and under-constraint

Expectations in Pip must avoid both over- and under-constraint. An over-constrained recognizer may be too strict and reject valid paths, while an under-constrained recognizer may accept invalid paths. Pip provides variant statements—repeats, xor, and futures—to allow the programmer to choose how specific to be in expressing expectations. Programmers should express how the system should behave rather than how it does behave, drawing upper and lower bounds and ordering constraints from actual program design.

Execution order is particularly prone to under- and over-constraint. For components that devote a thread to each request, asynchronous behavior is rare, and programmers will rarely, if ever, need to use futures. For event-based components, locks and communication order may impose constraints on event order, but there may be ambiguity. To deal with ambiguity, programmers should describe asynchronous tasks as futures. In particular, periodic background events (e.g., a timer callback) may require a future statement inside a repeat block, to allow many occurrences (perhaps an unknown number) at unknown times.

3.5 Implementation

The Pip trace checker operates as a nested loop: for each path instance in the trace, check it against each recognizer in the supplied expectations file.

Pip stores each recognizer as a list of thread patterns. Each thread pattern is a tree, with structure corresponding to the nested blocks in the expectations file. Figure 5 shows a sample expectation and one matching path. This example demonstrates why a greedy matching algorithm is insufficient to check expectations: the greedy algorithm would match Notice C too early and incorrectly return a match failure. Any correct matching algorithm must be able to check all possible sets of events that variants such as `maybe` and `repeat` can match.

Pip represents each path instance as a list of threads. Each thread is a tree, with structure corresponding to the hierarchy of tasks and subtasks. When checking a recognizer against a given path instance, Pip tries each thread in the path instance against each thread pattern in the recognizer. The recognizer matches the path instance if each path thread matches at least one thread pattern and each thread pattern matches an appropriate number of path threads.

Each type of expectation statement has a corresponding `check` function that matches path instance events. Each `check` function returns each possible number of events it could match. Literal statements (`task`, `notice`, `send`, and `recv`) match a single event, while variant statements (`repeat`, `xor`, and `any`) can match different numbers of events. For example, if two different branches of an `xor` statement could match, consuming either two or three events, `check` returns the set $[2, 3]$. If a literal statement matches the current path event, `check` returns $[1]$, otherwise $\emptyset$. When a `check` function for a variant statement returns $[0]$, it can be satisfied by matching zero events. A failure is indicated by the empty set, $\emptyset$.

The possible-match sets returned by each expectation statement form a search tree, with height equal to the number of expectation statements and width dependent on how many variant statements are present in the expectation. Pip uses a depth-first search to explore this search tree, looking for a leaf node that reaches the end

www.manaraa.com

```
task("A") {                          <task name="A">
    maybe { notice("B"); }               <notice name="B" />
    repeat between 1 and 2 {             <notice name="C" />
        notice(/.*/); }               </task>
    notice("C"); }
```

Figure 5: A sample fragment recognizer and a path that matches it.

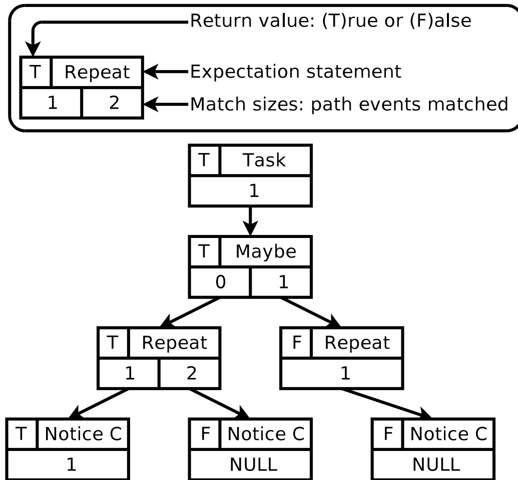

Figure 6: The search tree formed when matching the expectation and the path events in Figure 5.

```
future F1 { notice("C"); }           <task name="A">
task("A") {                              <notice name="B" />
    maybe { notice("B"); }               <notice name="C" />
    repeat between 1 and 2 {          </task>
        notice(/.*/); } }
```

Figure 7: The same path as in Figure 5, with a slightly modified recognizer to match it. Note that the `notice("C")` statement has been moved into a future block.

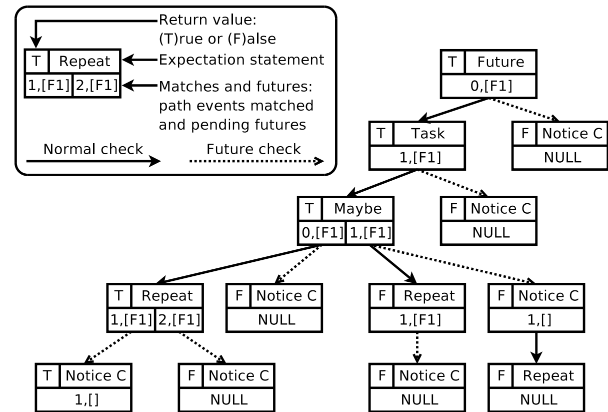

Figure 8: The search tree formed when matching the expectation and the path events in Figure 7.

of the expectation tree and the path tree at the same time. That is, the match succeeds if, in any branch of the search tree, the expectation matches all of the path events.

Figure 6 shows the possibilities searched when matching the expectations and the path events in Figure 5. Each node represents a `check` function call. Each node shows the return value (true or false) of the recursive search call, the expectation statement being matched, and the number(s) of events it can match. Leaves with no possible matches are shown with a possible-match set of `NULL` and a return value of false. A leaf with one or more possible matches might still return false, if any path events were left unmatched.

3.5.1 Futures

Pip checks futures within the same framework. Each `check` function takes an additional parameter containing a table of all currently available futures. Possible-match sets contain <events matched, futures table> tuples rather than just numbers of events that could be matched. Most `check` calls do not affect the table of active futures, simply returning the same value passed as a parameter. `Future.check` inserts a new entry into the futures table but does not attempt to match any events; it returns a single tuple: <0 events, updated futures table>. `Done.check` forces the named future to match immediately and removes it from the futures table.

Each node in the search tree must try all futures in the table as well as the next expectation statement. If a future matches, then that branch of the tree uses a new futures table with that one future removed. A leaf of the tree matches only if each expectation statement returns success, all path events are consumed, and the futures table is empty.

Figure 7 shows the same path instance as in Figure 5, with a different expectation to match it: the `notice("C")` statement is now a future. Figure 8 shows the possibilities searched when matching the expectations and the path events in Figure 7. Lazy evaluation again means that only a few nodes of the tree depicted in Figure 8 are actually expanded.

3.5.2 Performance

The time to load and check a path instance depends, of course, on the complexity of the path instance and the complexity of the recognizers Pip checks it against. On a 1.6 GHz laptop running Linux 2.6.13 and MySQL 4.1, a complex path instance containing 100 hosts and 1700 events takes about 12 ms to load and another 12 ms to check against seven recognizers, two of which contain futures. Thus, Pip can load and check about 40 complex path instances, or as many as 3400 simple path instances, per second on this hardware.

4 Annotations

Pip represents behavior as a list of path instances that contain tasks, notices, and messages, as described in Sec-

المنارة للاستشارات
www.manaraa.com

tion 2. These events are generated by source-code annotations. We chose annotations over other event and tracing approaches for two reasons. First, it was expedient. Our focus is expectations and how to generate, check, and visualize them automatically. Second, most other sources of events do not provide a path ID, making them less detailed and less accurate than annotations. Pip could easily be extended to incorporate any event source that provides path IDs.

Pip provides a library, libannotate, that programmers link into their applications. Programmers insert a modest number of source code annotations indicating which path is being handled at any given time, the beginning and end of interesting tasks, the transmission and receipt of messages, and any logging events relevant to path structure.

The six main annotation calls are:

- **annotate_set_path_id**(*id*): Indicate which path all subsequent events belong to. An application must set a path identifier before recording any other events. Path identifiers must be unique across all hosts and all time. Often, identifiers consist of the host address where the path began, plus a local sequence number.
- **annotate_start_task**(*name*): Begin some processing task, event handler, or subroutine. Annotation overhead for a task is around 10 μs, and the granularity for most resource measurements is a scheduler time slice. Thus, annotations are most useful for tasks that run for the length of a time slice or longer.
- **annotate_end_task**(*name*): End the given processing task.
- **annotate_send**(*id*, *size*): Send a message with the given identifier and size. Identifiers must be unique across all hosts and all time. Often, identifiers consist of the address of the sender, an indication of the type of message, and a local sequence number. Send events do not indicate the recipient address, allowing logical messages, anycast messages, forwarding, etc.
- **annotate_receive**(*id*, *size*): Receive a message with the given identifier and size. The identifier must be the same as when the message was sent, usually meaning that at least the sequence number must be sent in the message.
- **annotate_notice**(*string*): Record a log message.

Programs developed using a supported middleware layer may require only a few annotations. For example, we modified Mace [12], a high-level language for building distributed systems, to insert five of the six types of annotations automatically. Our modified mace adds begin- and end-task annotations for each transition (i.e., event handler), message-send and message-receive annotations for each network message and each timer, and set-path-id annotations before beginning a task or delivering a message. Only notices, which are optional and are the simplest of the six annotations, are left to the programmer. The programmer may choose to add further message, task, and path annotations beyond what our modified Mace generates.

Other middleware layers that handle event handling and network communication could automate annotations similarly. For example, we believe that SEDA [27] and RPC platforms like CORBA could generate message and task events and could propagate path IDs. Pinpoint [5] shows that J2EE can generate network and task events.

4.1 Reconciliation

The Pip annotation library records events in local trace files as the application runs. After the application terminates, the Pip *reconciler* gathers the files to a central location and loads them into a single database. The reconciler must pair start- and end-task events to make unified task events, and it must pair message-send and message-receive events to make unified message events.

The reconciler detects two types of errors. First, it detects incomplete (i.e., unpaired) tasks and messages. Second, it detects reused message IDs. Both types of errors can stem from annotation mistakes or from application bugs. In our experience, these errors usually indicate an annotation mistake, and they disappear entirely if annotations are added automatically.

5 Results

We applied Pip to several distributed systems, including FAB [25], SplitStream [4], Bullet [13, 15], and RanSub [14]. We found 18 bugs and fixed most of them. Some of the bugs we found affected correctness—for example, some bugs would result in SplitStream nodes not receiving data. Other bugs were pure performance improvements—we found places to improve read latency in FAB by 15% to 50%. Finally, we found correctness errors in SplitStream and RanSub that were masked at the expense of performance. That is, mechanisms intended to recover from node failures were instead recovering from avoidable programming errors. Using Pip, we discovered the underlying errors and eliminated the unnecessary time the protocols were spending in recovery code.

The bugs we found with Pip share two important characteristics. First, they occurred in actual executions of the systems under test. Pip can only check paths that are used in a given execution. Thus, path coverage is an important, though orthogonal, consideration. Second, the bugs manifested themselves through traced events. Program annotations must be comprehensive enough and expectations must be specific enough to isolate unexpected behavior. However, the bugs we found were not limited to bugs we knew about. That is, most of the bugs we

المنارة للاستشارات
www.manaraa.com

System	Lines of code	Recognizers (lines)	Lines of annotations	Number of hosts	Number of events	Trace duration	Reconciliation time (sec)	Checking time (sec)	Bugs found	Bugs fixed
FAB	124,025	17 (590)	28	4	88,054	4 sec	6	7	2	1
SplitStream	2,436	19 (983)	8	100	3,952,592	104 sec	1184	837	13	12
Bullet	2,447	1 (38)	23	100	863,197	71 sec	140	81	2	0
RanSub	1,699	7 (283)	32	100	312,994	602 sec	47	9	2	1

Table 2: System sizes, the effort required to check them, and the number of bugs found and fixed.

found were not visible when just running the applications or casually examining their log files.

Table 2 shows the size of each system we tested, along with how much programmer effort and CPU time it took to apply Pip in each case. Bullet has fewer expectations because we did not write validators for all types of Bullet paths. SplitStream has many expectations because it is inherently complex and because in some cases we wrote both a validator and an overly general recognizer for the same class of behavior (see Section 5.2). Over 90% of the running time of reconciliation and checking is spent in MySQL queries; a more lightweight solution for storing paths could yield dramatic speed improvements. In addition to the manual annotations indicated in the table, we added 55 annotation calls to the Mace compiler and library and 19 to the FAB IDL compiler.

Reconciliation time is $O(E \lg p)$ for E events and p path instances, as each event is stored in a database, indexed by path ID. The number of high-level recognizer checking operations is exactly rp for p path instances and r recognizers. Neither stage's running time is dependent on the number of hosts or on the concurrency between paths. The checking time for a path instance against a recognizer is worst-case exponential in the length of the recognizer, e.g., when a recognizer with pathologically nested future and variant statements almost matches a given path instance. In practice, we did not encounter any recognizers that took more than linear time to check.

In the remainder of this section, we will describe our experiences with each system, some sample bugs we found, and lessons we learned.

5.1 FAB

A Federated Array of Bricks (FAB) [25] is a distributed block storage system built from commodity Linux PCs. FAB replicates data using simple replication or erasure coding and uses majority voting protocols to protect against node failures and network partitions. FAB contains about 125,000 lines of C++ code and a few thousand lines of Python. All of FAB's network code is automatically generated from IDL descriptions written in Python. The C++ portions of FAB combine user-level threading and event-handling techniques. A typical FAB configuration includes four or more hosts, background membership and consensus communication, and a mix of concurrent read and write requests from one or more clients.

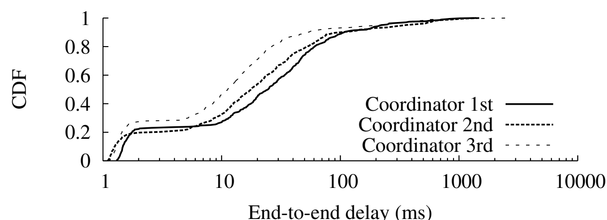

Figure 9: CDF of end-to-end latency in milliseconds for FAB read operations. The left-most line shows the case where the coordinator calls itself last. Note that the x axis is log-scaled to show detail.

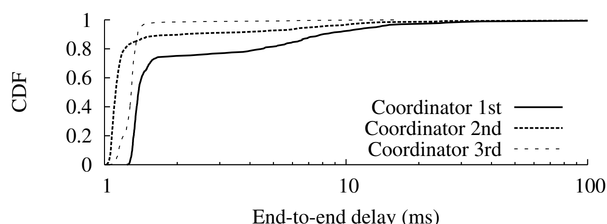

Figure 10: CDF of end-to-end latency in milliseconds for FAB read operations in a system with a high cache hit rate. The left-most line shows the case where the coordinator calls itself second. Note that the x axis is log-scaled to show detail.

We were not initially familiar with FAB, but we had access to its source code, and one of its authors offered to help us understand it. With just a few hours of effort, we annotated FAB's IDL compiler, and were able to get the tasks and messages necessary to examine every protocol.

Figure 3 in Section 3.3 showed an expectation for the FAB read protocol when the node coordinating the access (the I/O coordinator) does not contain a replica of the block requested. In this section, we focus on the case where the coordinator does contain a replica. In addition to the read and write protocols, we annotated and wrote expectations for FAB's implementation of Paxos [17] and the Cristian-Schmuck membership protocol [6] but did not find any bugs in either.

Bugs: When the FAB I/O coordinator contains a replica of the block requested, the order of RPCs issued affects performance. In FAB, an RPC issued by a node to itself is handled synchronously. Originally, FAB issued read or write RPCs to all replicas in an arbitrary order. A recent optimization changed this code so that the coordinator always issues the RPC to itself (if any) last, allowing greater overlap of computation.

المنارة للاستشارات
www.manaraa.com

FAB's author sent us the unoptimized code without describing the optimization to us, with the intention that we use Pip to rediscover the same optimization. Figure 9 shows the performance of read operations when the coordinator calls itself first, second, or last. When the block is not in cache (all delay values about 10 ms), having the coordinator issue an RPC to itself last is up to twice as fast as either other order. Write performance shows a similar, though less pronounced, difference.

We discovered this optimization using expectations and the visualization GUI together. We wrote recognizers for the cases where the coordinator called itself first, second, and third and then graphed several properties of the three path sets against each other. The graph for end-to-end delay showed a significant discrepancy between the coordinator-last case and the other two cases.

Figure 10 shows the same measurements as Figure 9, in a system with a higher cache hit rate. We noticed that letting the coordinator call itself second resulted in a 15% decrease in latency for reads of cached data by performing the usually unneeded third call after achieving a 2-of-3 quorum and sending a response to the client. The FAB authors were not aware of this difference.

Lessons: Bugs are best noticed by someone who knows the system under test. We wrote expectations for FAB that classified read and write operations as valid regardless of the order of computation. We found it easy to write recognizers for the actual behavior a system exhibits, or even to generate them automatically, but only someone familiar with the system can say whether such recognizers constitute real expectations.

5.2 SplitStream

SplitStream [4] is a high-bandwidth content-streaming system built upon the Scribe multicast protocol [24] and the Pastry DHT [23]. SplitStream sends content in parallel over a "forest" of 16 Scribe trees. At any given time, SplitStream can accommodate nodes joining or leaving, plus 16 concurrent multicast trees. We chose to study SplitStream because it is a complex protocol, we have an implementation in Mace, and our implementation was exhibiting both performance problems and structural bugs. Our SplitStream tests included 100 hosts running under ModelNet [26] for between two and five minutes.

Bugs: We found 13 bugs in SplitStream and fixed most of them. Space does not allow descriptions of all 13 bugs. We found two of the bugs using the GUI and 11 of the bugs by either using or writing expectations. Seven bugs had gone unnoticed or uncorrected for ten months or more, while the other six had been introduced recently along with new features or as a side effect of porting SplitStream from MACEDON to Mace. Four of the bugs we found were due to an incorrect or incomplete understanding of the SplitStream protocol, and the other nine were implementation errors. At least four of the bugs resulted in inefficient (rather than incorrect) behavior. In these cases, Pip enabled performance improvements by uncovering bugs that might have gone undetected in a simple check of correctness.

One bug in SplitStream occurred twice, with similar symptoms but two different causes. SplitStream allows each node to have up to 18 children, but in some cases was accepting as many as 25. We first discovered this bug using the GUI: visualizations of multicast paths' causal structure sometimes showed nodes with too many children. The cause the first time was the use of global and local variables with the same name; SplitStream was passing the wrong variable to a call intended to offload excess children. After fixing this bug, we wrote a validator to check the number of children, and it soon caught more violations. The second cause was an unregistered callback. SplitStream contains a function to accept or reject new children, but the function was never called.

Lessons: Some bugs that look like structural bugs affect only performance, not correctness. For example, when a SplitStream node has too many children, the tree still delivers data, but at lower speeds. The line between structural bugs and performance bugs is not always clear.

The expectations checker can help find bugs in several ways. First, if we have an expectation we know to be correct, the checker can flag paths that contain incorrect behavior. Second, we can generate recognizers automatically to match existing paths. In this case, the recognizer is an external description of actual behavior rather than expected behavior. The recognizer is often more concise and readable than any other summary of system behavior, and bugs can be obvious just from reading it. Finally, we can write an overly general recognizer that matches all multicast paths and a stricter, validating recognizer that matches only correct multicast paths. Then we can study incorrect multicast paths—those matched by the first but not the second—without attempting to write validators for other types of paths in the system.

5.3 Bullet

Bullet [13, 15] is a third-generation content-distribution mesh. Unlike overlay multicast protocols, Bullet forms a mesh by letting each downloading node choose several peers, which it will send data to and receive data from. Peers send each other lists of which blocks they have already received. One node can decide to send (push) a list of available blocks to its peers, or the second can request (pull) the list. Lists are transmitted as deltas containing only changes since the last transmission between the given pair of nodes.

Bugs: We found two bugs in Bullet, both of which are inefficiencies rather than correctness problems. First, a

www.manaraa.com

given node A sometimes notifies node B of an available block N several times. These extra notifications are unexpected behavior. We found these extra notifications using the reconciler rather than the expectations checker. We set each message ID as <sender, recipient, block number> instead of using sequence numbers. Thus, whenever a block notification is re-sent, the reconciler generates a "reused message ID" error.

The second bug is that each node tells each of its peers about every available block, even blocks that the peers have already retrieved. This bug is actually expected behavior, but in writing expectations for Pip we realized it was inefficient.

Lessons: We were interested in how notifications about each block propagate through the mesh. Because some notifications are pulls caused by timers, the propagation path is not causal. Thus, we had to write additional annotations for *virtual* paths in addition to the causal paths that Mace annotated automatically.

Pip can find application errors using the reconciler, not just using the path checker or the GUI. It would have been easy to write expectations asserting that no node learns about the same block from the same peer twice, but it was not necessary because the reconciler flagged such repeated notifications as reused message IDs.

5.4 RanSub

RanSub [14] is a building block for higher-level protocols. It constructs a tree and tells each node in the tree about a uniformly random subset of the other nodes in the tree. RanSub periodically performs two phases of communication: *distribute* and *collect*. In the distribute phase, each node starting with the root sends a random subset to each of its children. In the collect phase, each node starting with the leaves sends a summary of its state to its parent. Interior nodes send a summary message only after receiving a message from all children. Our RanSub tests involved 100 hosts and ran for 5 minutes.

Because RanSub is written in Mace, we were able to generate all needed annotations automatically.

Bugs: We found two bugs in RanSub and fixed one of them. First, each interior node should only send a summary message to its parent after hearing from all of its children. Instead, the first time the collect phase ran, each interior node sent a summary message after hearing from one child. We found this bug by writing an expectation for the collect-and-distribute path; the first round of communication did not match. The root cause was that interior nodes had some state variables that did not get initialized until after the first communication round. We fixed this bug.

The second bug we found in RanSub is a performance bug. The end-to-end latency for collect-and-distribute

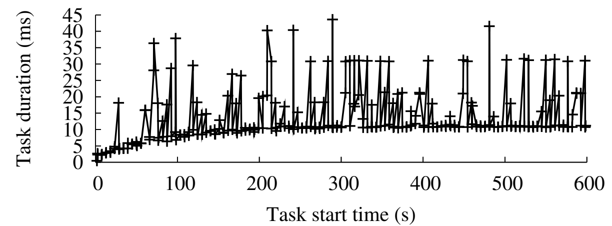

Figure 11: Duration for the `deliverGossip` task as a function of time.

paths starts out at about 40 ms and degrades gradually to about 50 ms after running for three minutes. We traced the bottleneck to a task called `deliverGossip` that initially takes 0 ms to run and degrades gradually to about 11 ms. We found this bug using the GUI. First, we examined the end-to-end latency as a function of time. Seeing an error there, we checked each class of tasks in turn until we found the gossip task responsible for the degradation. Figure 11 shows the time consumed by the gossip task as a function of time. The reason for `deliverGossip` degrading over time is unclear but might be that `deliverGossip` logs a list of all gossip previously received.

6 Related work

Pip is one of many approaches to finding structure and performance bugs in distributed systems. Below, we highlight two categories of debugging approaches: path analysis tools and automated expectation checking. Pip is the first to combine the two approaches. Finally, we discuss the relationship between Pip and high-level languages for specifying and developing distributed systems.

6.1 Path analysis tools

Several previous systems have modeled the behavior of distributed systems as a collection of causal paths. This approach is particularly appropriate for systems driven by user requests, as it captures the delays and resource consumption associated with each request. Path-based debugging can enable programmers to find aberrant paths and to optimize both throughput and end-to-end latency.

Project 5 [1] infers causal paths from black-box network traces. By doing so, it can help debug systems with unavailable source code. Deploying black-box debugging, at least in theory, requires less effort than annotating source code. However, Project 5 can only report what it can infer. Its granularity is limited to host-to-host communication, and it often reconstructs paths incorrectly. In particular, interesting paths, including infrequent paths or paths with long or variable delays, may be lost.

Magpie [2] reconstructs causal paths based on OS-level event tracing. Like Project 5, Magpie can oper-

www.manaraa.com

ate without access to source code. However, Magpie can construct paths with much higher accuracy than Project 5 can, because OS-level tracing provides more information than network tracing alone. Magpie clusters causal paths using a string-edit-distance algorithm and identifies outliers—that is, small clusters.

Like Pip, Pinpoint [5] constructs causal paths by annotating applications or platforms to generate events and maintain a unique path identifier per incoming request. Like Pip and Magpie, Pinpoint can construct paths with a high degree of confidence because it does not rely on inference. Like Magpie but unlike Pip, Pinpoint assumes that anomalies indicate bugs. Pinpoint uses a probabilistic, context-free grammar to detect anomalies on a per-event basis rather than considering whole paths. Doing so significantly underconstrains path checking, which, as the authors point out, may cause Pinpoint to validate some paths with bugs.

All three of these existing causal path debugging systems rely on statistical inference to find unusual behavior and assume that unusual behavior indicates bugs. Doing so has two drawbacks. First, inference requires large traces with many path instances. Second, these systems can all miss bugs in common paths or incorrectly identify rare but valid paths.

The accuracy and granularity of existing causal path debugging tools are limited by what information they can get from traces of unmodified applications. In practice, these systems entail a form of gray-box debugging, leveraging prior algorithmic knowledge, observations, and inferences to learn about the internals of an unmodifiable distributed system. In contrast, Pip assumes the ability to modify the source for at least parts of a distributed system, and it provides richer capabilities for exploring systems without prior knowledge and for automatically checking systems against high-level expectations.

6.2 Automated expectation checking

Several existing systems support expressing and checking expectations about structure or performance. Some of the systems operate on traces, others on specifications, and still others on source code. Some support checking performance, others structure, and others both. Some, but not all, support distributed systems.

PSpec [21] allows programmers to write assertions about the performance of systems. PSpec gathers information from application logs and runs after the application has finished running. The assertions in PSpec all pertain to the performance or timing of intervals, where an interval is defined by two events (a start and an end) in the log. PSpec has no support for causal paths or for application structure in general.

Meta-level compilation (MC) [8] checks source code for static bugs using a compiler extended with system-specific rules. MC checks all code paths exhaustively but is limited to single-node bugs that do not depend on dynamic state. MC works well for finding the root causes of bugs directly, while Pip detects symptoms and highlights code components that might be at fault. MC focuses on individual incorrect statements, while Pip focuses on the correctness of causal paths, often spanning multiple nodes. MC finds many false positives and bugs with no effect, while Pip is limited to actual bugs present in a given execution of the application.

Paradyn [19] is a performance measurement tool for complex parallel and distributed software. The Paradyn Configuration Language (PCL) allows programmers to describe expected characteristics of applications and platforms, and in particular to describe metrics; PCL seems somewhat analogous to Pip's expectation language. However, PCL cannot express the causal path structure of threads, tasks and messages in a program, nor does Paradyn reveal the program's structure.

6.3 Domain-specific languages

Developers of distributed systems have a wide variety of specification and implementation languages to choose from. Languages like Estelle [11], π-calculus [20], join-calculus [9], and P2 [18] embrace a formal, declarative approach. Erlang [3] and Mace [12] use an imperative approach, with libraries and language constructs specialized for concurrency and communication. Finally, many programmers still use traditional, general-purpose languages like Java and C++.

Pip is intended primarily for developers using imperative languages, including both general-purpose languages and domain-specific languages for building distributed systems. Pip provides language bindings for Java, C, C++, and Mace. While programmers using declarative languages can verify the correctness of their programs through static analysis, Pip is still valuable for monitoring and checking dynamic properties of a program, such as latency, throughput, concurrency, and node failure.

7 Conclusions

Pip helps programmers find bugs in distributed systems by comparing actual system behavior to the programmer's expectations about that behavior. Pip provides visualization of expected and actual behavior, allowing programmers to examine behavior that violates their expressed expectations, and to search interactively for additional unexpected behavior. The same techniques can help programmers learn about an unfamiliar system or monitor a deployed system.

Pip can often generate any needed annotations automatically, for applications constructed using a supported middleware layer. Pip can also generate initial expectations automatically. These generated expectations are

المنارة للاستشارات
www.manaraa.com

often the most readable description of system behavior, and bugs can be obvious just from reading them.

We applied Pip to a variety of distributed systems, large and small, and found bugs in each system.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. SOSP*, Bolton Landing, NY, Oct. 2003.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modeling. In *Proc. OSDI*, San Francisco, CA, Dec. 2004.

[3] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. Core Erlang 1.0 language specification. Technical Report 030, Uppsala University, Nov. 2000.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth multicast in cooperative environments. In *Proc. SOSP*, Bolton Landing, NY, Oct. 2003.

[5] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *Proc. NSDI*, San Francisco, CA, April 2004.

[6] F. Cristian and F. Schmuck. Agreeing on processor group membership in timed asynchronous distributed systems. Report CSE95-428, UC San Diego, 1995.

[7] C. Dickens. *Great Expectations*. Chapman & Hall, London, 1861.

[8] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. OSDI*, San Diego, CA, Dec. 2000.

[9] C. Fournet and G. Gonthier. The join calculus: a language for distributed mobile programming. In *Proc. APPSEM*, Caminha, Portugal, 2000.

[10] P. Godefroid. Software model checking: the VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, Mar. 2005.

[11] ISO 9074. Estelle: A formal description technique based on an extended state transition model. 1987.

[12] Mace. http://mace.ucsd.edu, 2005.

[13] D. Kostić, R. Braud, C. Killian, E. Vandekieft, J. W. Anderson, A. C. Snoeren, and A. Vahdat. Maintaining high bandwidth under dynamic network conditions. In *Proc. USENIX 2005*, Anaheim, CA, Apr. 2005.

[14] D. Kostić, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using random subsets to build scalable network services. In *Proc. USITS*, Seattle, WA, Mar. 2003.

[15] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. SOSP*, Bolton Landing, NY, Oct. 2003.

[16] L. Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, May 1994.

[17] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2):133–169, May 1998.

[18] B. T. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proc. SOSP*, Brighton, UK, Oct. 2005.

[19] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, Nov. 1995.

[20] R. Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, Oct. 1991.

[21] S. E. Perl and W. E. Weihl. Performance assertion checking. In *Proc. SOSP*, Asheville, NC, Dec. 1993.

[22] A. Rodriguez, C. Killian, S. Bhat, D. Kostić, and A. Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *Proc. NSDI*, San Francisco, CA, April 2004.

[23] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Proc. Middleware'2001*, Heidelberg, Germany, Nov. 2001.

[24] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The Design of a Large-scale Event Notification Infrastructure. In *3rd Intl. Workshop on Networked Group Communication*, London, UK, Nov. 2001.

[25] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *Proc. ASPLOS*, Boston, MA, 2004.

[26] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proc. OSDI*, Boston, MA, 2002.

[27] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proc. SOSP*, Banff, Canada, 2001.

Notes

[1] Pip is the main character in *Great Expectations* [7].

[2] Source code and screenshots for Pip are available at `http://issg.cs.duke.edu/pip`.

[3] Mace is an ongoing redesign of the MACEDON [22] language for building distributed systems.

المنارة للاستشارات
www.manaraa.com